Il Client Bitcoin

Bitcoin Core: L'Implementazione di Riferimento

Puoi scaricare il client di riferimento *Bitcoin Core*, conosciuto anche come "Satoshi client" da bitcoin.org. Il client di riferimento implementa tutti gli aspetti del sistema bitcoin, questo include i wallet, un motore di verifica delle transazioni con una copia completa dell'intero registro delle transazioni (blockchain) e un nodo completo nella rete peer-to-peer bitcoin.

Su Pagina Scegli il tuo Wallet del sito Bitcoin org, seleziona Bitcoin Core per scaricare il client di riferimento. A seconda del tuo sistema operativo, scaricherai un'installer eseguibile. Per windows, questo sarà un archivio ZIP o un'eseguibile .exe. Per Mac OS è un'immagine disco .dmg. Le versioni di Linux includono un pacchetto PPA per ubuntu o un archivio tar .gz. La pagina bitcoin.org che elenca i client bitcoin consigliati è mostrata in Scegliendo un client bitcoin su bitcoin.org

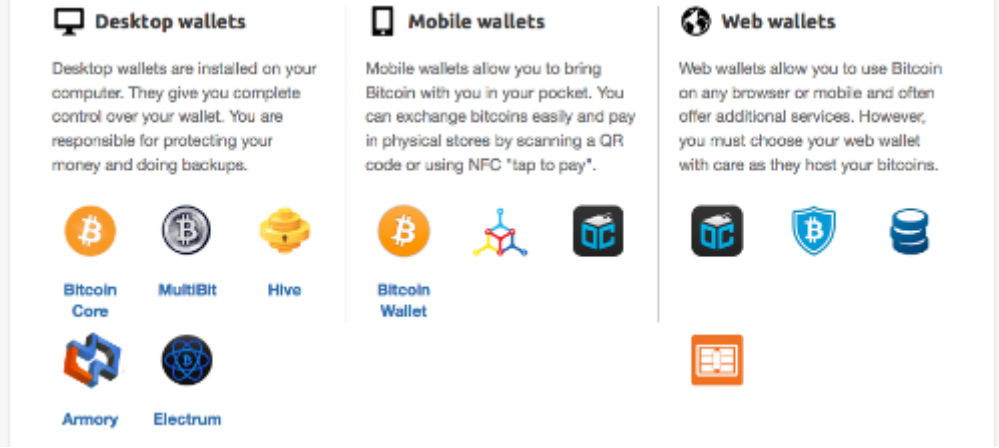

Figure 1. Scegliendo un client bitcoin su bitcoin.org

Eseguire Bitcoin Core per la Prima Volta

Se stai scaricando un pacchetto installabile, come un .exe, .dmg, o PPA, potrai installarle nello stesso modo un'applicazione sul tuo sistema operativo. Per Windows, esegui l'.exe e segui le istruzioni passo-per-passo. Per Mac OS, lancia il .dmg e sposta l'icona di Bitcoin-QT nella tua cartella *Applicazioni*. Per Ubuntu, clicca due volte sul PPA nel tuo File Explorer e questo aprirà il package manager per installare il pacchetto. Una volta completata l'installazione dovresti avere una nuova applicazione chiamata Bitcoin-QT tra la lista delle tue applicazioni attualmente installate. Clicca due volte l'icona per avviare il client bitcoin.

La prima volta quando esegui Bitcoin Core, inizierà a scaricare la blockchain, un processo che potrebbe impiegare qualche giorno (vedi Schermata di Bitcoin Core durante l'inizializzazione della blockchain). Lascialo girare in background fino a che non mostrerà "Synchronized" (Sincronizzato) e non "out of sync" accanto al balance (saldo).

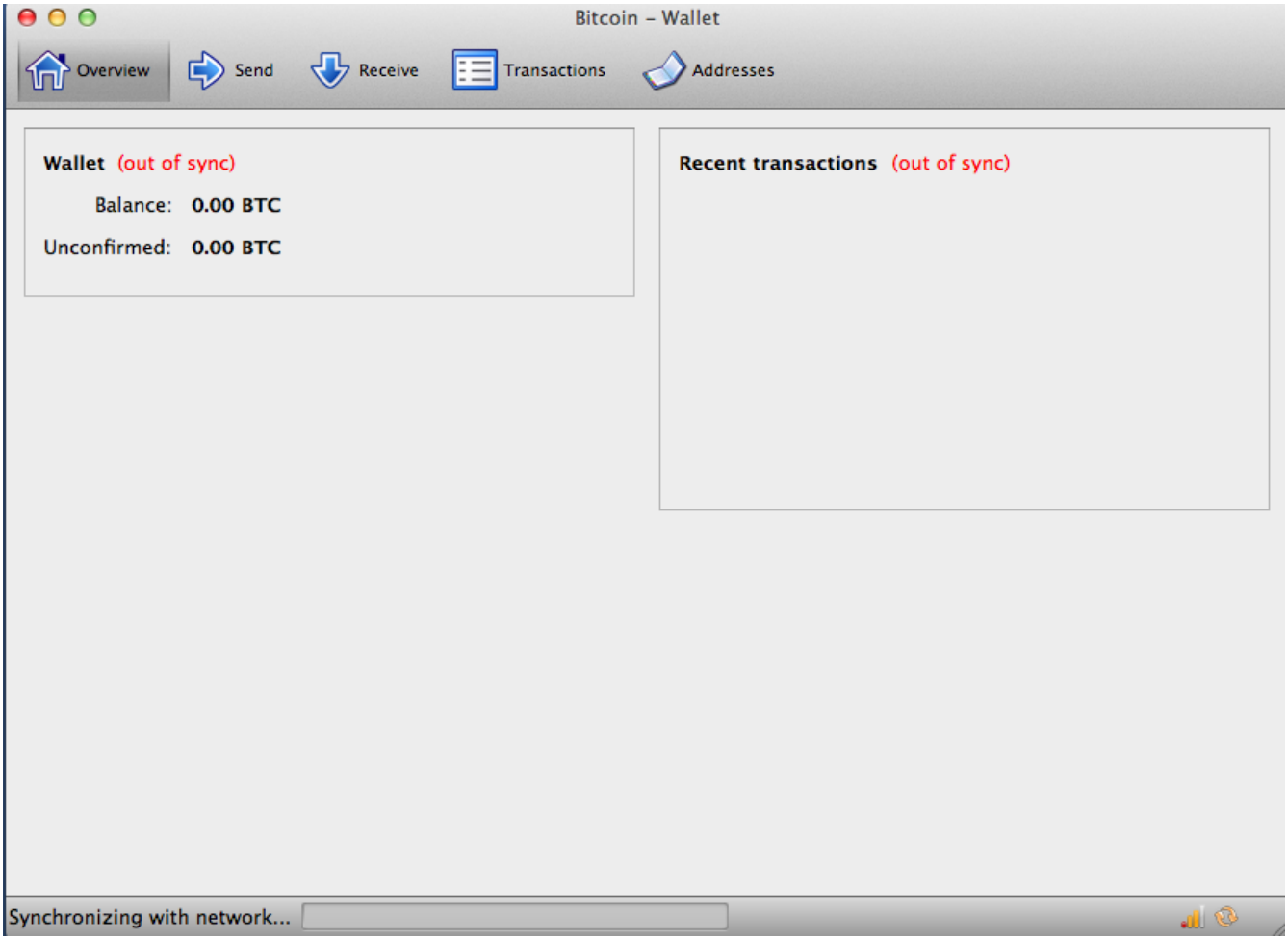

Figure 2. Schermata di Bitcoin Core durante l'inizializzazione della blockchain

TIP

Bitcoin Core mantiene una copia completa di tutto il registro delle transazioni (blockchain), con ogni transazione che sia mai avvenuta nel network bitcoin fin dalla sua nascita nel 2009. Questo dataset ha una dimensione di svariati gigabytes (approssimativamente 16 GB a fine 2013) ed è scaricato in modo incrementale nell'arco di qualche giorno. Il client non sarà in grado di processare transazioni o aggiornare i saldo degli account fino a che il dataset della blockchain non sia stato scaricato completamente. Durante questo periodo, il client mostrerà il messaggio "out of sync" (non sincronizzato) vicino al saldo dei conti e "Synchronizing" (sincronizzazione in corso) nel footer. Assicurati di avere sufficiente spazio disco, una buona connessione e tempo per completare la sincronizzazione iniziale.

Compilare Bitcoin Core dal Codice Sorgente

Per gli sviluppatori, c'è anche l'opzione di scaricare il codice sorgente completo come un file ZIP o clonare il sorgente principale dal repository presente su GitHub. Sulla pagina bitcoin/bitcoin su GitHub, seleziona Download ZIP dalla barra in alto. In alternativa, utilizza il comando git da terminale per creare una copia locale del codice sorgente sulla tua macchina. Nell'esempio seguente, stiamo clonando il codice sorgente da riga di comando Unix, presente in Linux e Mac OS.

```
$ git clone https://github.com/bitcoin/bitcoin.git
Cloning into 'bitcoin'...
remote: Counting objects: 31864, done.
remote: Compressing objects: 100% (12007/12007), done.
remote: Total 31864 (delta 24480), reused 26530 (delta 19621)
Receiving objects: 100% (31864/31864), 18.47 MiB | 119 KiB/s, done.
Resolving deltas: 100% (24480/24480), done.
$
```

TIP Le istruzioni e l'output ritornato potrebbero variare da una versione all'altra. Segui la documentazione fornita con il codice anche se differisce dalle istruzioni riportate qui, e non sorprenderti se l'output mostrato sul tuo schermo è leggermente differente dagli esempi riportati di seguito.

Non appena l'operazione del comando git clone sarà completata, avrai una copia locale completa del repository del codice nella directory *bitcoin*. Entra dentro questa directory digitando cd bitcoin nel prompt/terminale:

```
$ cd bitcoin
```

Di default, la copia locale verrà sincronizzata con il codice più recente, che potrebbe essere una versione beta o unsable di bitcoin. Prima di compilare il codice, seleziona una versione specifica facendo il checkout di un release *tag*. Questo farà sincronizzare la copia locale con la versione specifica del codice del repository identificato con un'etichetta detta tag. I tag sono usati dai developer per segnare release specifiche del codice per numero di versione. Per prima cosa, per trovare la lista di tutti i tag disponibili, utilizzeremo il comando git tag:

```
$ git tag
v0.1.5
v0.1.6test1
v0.2.0
v0.2.10
v0.2.11
v0.2.12

[... molti altri tag ...]

v0.8.4rc2
v0.8.5
v0.8.6
v0.8.6rc1
v0.9.0rc1
```

La lista dei tag mostra tutte le versioni di bitcoin rilasciate. Per convenzione, le *release candidate*, che sono fatte per testing, hanno il suffisso "rc". Le release stabili (stable) che possono essere eseguite su sistemi in produzione non hanno suffisso. Dalla lista precedente, seleziona la release

maggiore che al momento della scrittura di questo libro è la v0.9.0rc1. Per sincronizzare il codice locale con questa versione, utilizza il comando git checkout:

```
$ git checkout v0.9.0rc1
Note: checking out 'v0.9.0rc1'.

HEAD is now at 15ec451... Merge pull request #3605
$
```

Il codice sorgente include la documentazione, che può essere trovata in molti file. Controlla la documentazione principale che si trova nel *README.md* nella directory bitcoin digitando more README.md nel prompt e usando la barra spaziatrice per procedere alla pagina successiva. In questo capitolo, compileremo il bitcoin client da riga di comando, anche conosciuto come bitcoind su Linux. Ricontrolla le istruzioni per compilare il client da riga di comando bitcoind per la tua distribuzione digitando more doc/build-unix.md. Istruzioni alternative per Mac OS X e Windows sono disponibili nella directory *doc*, rispettivamente in *build-osx.md* o *build-msw.md*.

Revisiona attentamente i requisiti di build, che sono nella prima parte della documentazione della build. Queste sono le librerie che devono essere presenti sul tuo sistema prima che tu possa iniziare a compilare bitcoin. Se questi prerequisiti non sono presenti, il processo di build fallirà con un errore. Se questo avviene perchè ti manca un prerequisito, puoi installarlo e poi continuare il processo di build da dove eri rimasto. Assumendo che i prerequisiti siano installati, inizierai il processo di build generando una serie di script di build usando lo script *autogen.sh*.

TIP

Il processo di build di Bitcoin Core è stato modificato perchè utilizzi il sistema di autogen/configure/make dalla versione 0.9. Versioni precedenti utilizzavano un semplice Makefile e funzionavano in modo leggermente differente dal seguente esempio. Segui le istruzioni per la versione che vuoi compilare. L'autogen/configure/make introdotto nella 0.9 sarà probabilmente il build system utilizzato per tutte le versioni future del codice e è il sistema mostrato nei seguenti esempi.

```
$ ./autogen.sh
configure.ac:12: installing `src/build-aux/config.guess'
configure.ac:12: installing `src/build-aux/config.sub'
configure.ac:37: installing `src/build-aux/install-sh'
configure.ac:37: installing `src/build-aux/missing'
src/Makefile.am: installing `src/build-aux/depcomp'
$
```

Lo script *autogen.sh* crea una serie di script di configurazione automatici che interrogheranno il tuo sistema per scoprire i settaggi corretti e che assicureranno che tu abbia tutte le librerie necessarie per compilare il codice. La cosa più importante di queste è che lo script di configure offre una serie di opzioni differenti per personalizzare il processo di build. Digita ./configure --help per vedere tutte le opzioni disponibili:

```
$ ./configure --help

`configure' configures Bitcoin Core 0.9.0 to adapt to many kinds of systems.
('configure' configura Bitcoin Core 0.9.0 per adattarlo a diversi tipi di sistemi.)

Usage (Utilizzo): ./configure [OPTION]... [VAR=VALUE]...

To assign environment variables (e.g., CC, CFLAGS...), specify them as (Per assegnare
variabili d'ambiente - specificale come)
VAR=VALUE.  See below for descriptions of some of the useful variables. (Vedi di
seguito per una descrizione delle variabili utili.)

Defaults for the options are specified in brackets.

Configuration:
  -h, --help              display this help and exit (mostra questo messaggio di aiuto
e esce)
      --help=short        display options specific to this package (mostra le opzioni
specifiche di questo pacchetto)
      --help=recursive    display the short help of all the included packages (mostra
il testo di aiuto di tutti i pacchetti inclusi)
  -V, --version           display version information and exit (mostra le informazioni
di versione)

[... moltre altre opzioni e variabili saranno mostrate di seguente ...]

Funzionalità Opzionali:
  --disable-option-checking  ignore unrecognized --enable/--with options
  --disable-FEATURE       do not include FEATURE (same as --enable-FEATURE=no) (non
includere la funzionalità FEATURE)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]

[... altre opzioni ...]

Utilizza queste variabili per sovrascrivere le scelte fatte da `configure` o per
facilitare
la possibilità di caricare librerie e programmi con nomi/percorsi non standard.

Comunica i bug a <info@bitcoin.org>.

$
```

Lo script configure permette di abilitare o disabilitare alcune funzioni di bitcoind attraverso l'uso dei flag --enable-FEATURE e --disable-FEATURE, nei quali FEATURE è sostituita dal nome della funzionalità da abilitare/disabilitare, come listato nell'output del comando help. In questo capitolo, compileremo il client bitcoind con tutte le funzionalità di base. Non useremo i flag di configurazione, ma dovresti ricontrollarli per capire quali funzionalità opzionali fanno parte del client. Poi, esegui lo script configure per scoprire automaticamente tutte le librerie necessarie e per creare uno script di compilazione customizzato per il tuo sistema:

```
$ ./configure
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes

[... molte altre funzionalità del sistema sono testate ...]

configure: creating ./config.status
config.status: creating Makefile
config.status: creating src/Makefile
config.status: creating src/test/Makefile
config.status: creating src/qt/Makefile
config.status: creating src/qt/test/Makefile
config.status: creating share/setup.nsi
config.status: creating share/qt/Info.plist
config.status: creating qa/pull-tester/run-bitcoind-for-test.sh
config.status: creating qa/pull-tester/build-tests.sh
config.status: creating src/bitcoin-config.h
config.status: executing depfiles commands
$
```

Se tutto va bene, il comando configure terminerà creando gli script di build personalizzati che ci permetteranno di compilare bitcoind. Se qualche libreria è mancante o se ci sono errori, il comando configure terminerà con un errore invece di creare gli script di build. Se un errore viene riportato, sarà probabilmente perchè ti manca una libreria o c'è una libreria non compatibile. Ricontrolla nuovamente la documentazione di build e sii sicuro di aver installato tutti i prerequisiti mancanti. A questo punto lancia configure nuovamente e ricontrolla se l'errore sia stato risolto. Infine, compilerai il codice sorgente, un processo che potrà impiegare fino a un'ora per essere completato. Durante il processo di compilazione potrai vedere output entro pochi secondi o entro pochi minuti, o un'errore se qualcosa va storto. Il processo di compilazione può essere avviato nuovamente in qualsiasi momento se per qualche ragione viene interrotto. Digita make per iniziare a compilare:

```
$ make
Making all in src
make[1]: Entering directory `/home/ubuntu/bitcoin/src'
make  all-recursive
make[2]: Entering directory `/home/ubuntu/bitcoin/src'
Making all in .
make[3]: Entering directory `/home/ubuntu/bitcoin/src'
  CXX    addrman.o
  CXX    alert.o
  CXX    rpcserver.o
  CXX    bloom.o
  CXX    chainparams.o

[... seguono molti altri messaggi di compilazione ...]

  CXX    test_bitcoin-wallet_tests.o
  CXX    test_bitcoin-rpc_wallet_tests.o
  CXXLD  test_bitcoin
make[4]: Leaving directory `/home/ubuntu/bitcoin/src/test'
make[3]: Leaving directory `/home/ubuntu/bitcoin/src/test'
make[2]: Leaving directory `/home/ubuntu/bitcoin/src'
make[1]: Leaving directory `/home/ubuntu/bitcoin/src'
make[1]: Entering directory `/home/ubuntu/bitcoin'
make[1]: Nothing to be done for `all-am'.
make[1]: Leaving directory `/home/ubuntu/bitcoin'
$
```

Se tutto è andato bene, bitcoind dovrebbe essere compilato. Il passo finale a questo punto è installare l'eseguibile bitcoind nel sistema usando il comando make:

```
$ sudo make install
Making install in src
Making install in .
 /bin/mkdir -p '/usr/local/bin'
  /usr/bin/install -c bitcoind bitcoin-cli '/usr/local/bin'
Making install in test
make  install-am
 /bin/mkdir -p '/usr/local/bin'
  /usr/bin/install -c test_bitcoin '/usr/local/bin'
$
```

Puoi confermare che bitcoin è correttamente installato chiedendo al sistema il percorso dei due eseguibili, come di seguito:

```
$ which bitcoind
/usr/local/bin/bitcoind

$ which bitcoin-cli
/usr/local/bin/bitcoin-cli
```

L'installazione di bitcoind di default lo metterà in */usr/local/bin*. Quando lancerai bitcoind per la prima volta, ti ricorderà di creare un file di configurazione con una strong password (una password abbastanza complessa e quindi sicura) per l'interfaccia JSON-RPC. Esegui bitcoind digitando bitcoind nel terminale:

```
$ bitcoind
Errore: Per utilizzare l'opzione "-server", dovrai impostare una rpcpassword nel file
di configurazione:
/home/ubuntu/.bitcoin/bitcoin.conf
E' consigliato usare la seguente password casuale:
rpcuser=bitcoinrpc
rpcpassword=2XA4DuKNCbtZXsBQRRNDEwEY2nM6M4H9Tx5dFjoAVVbK
(non c'è bisogno che ti ricordi questa password)
L'username e la password NON DEVONO essere uguali.
Se il file non esiste, crealo con i permessi owner-readable-only (leggibile solo dal
proprietario).
E' inoltre consigliato impostare alertnotify di modo da essere notificato riguardo
eventuali problemi;
per esempio: alertnotify=echo %s | mail -s "Bitcoin Alert" admin@foo.com
```

Modifica il file di configurazione nel tuo editor preferito e imposta i parametri, sostituisci la password con una password sicura (strong password) come ti raccomandano le istruzioni di bitcoind. *Non* usare la password mostrata di seguito. Crea il file dentro la directory *.bitcoin* in modo che il percorso finale sia *.bitcoin/bitcoin.conf* e imposta un'username e una password:

```
rpcuser=bitcoinrpc
rpcpassword=2XA4DuKNCbtZXsBQRRNDEwEY2nM6M4H9Tx5dFjoAVVbK
```

Mentre stai modificando questo file di configurazione, probabilmente vorrai impostare alcune opzioni in più, come txindex (vedi Indice del Database delle Transazioni e Opzione txindex). Per una lista completa delle opzioni disponibili, digita bitcoind --help.

A questo punto, esegui il client Bitcoin Core. La prima volta che verrà eseguito, ricostruirà la blockchain di bitcoin scaricando tutti i blocchi. Questo è un file di molti gigabyte e richiederà una media di tre giorni per essere completamente scaricato. Potrai accorciare il tempo di inizializzazione della blockchain scaricando una copia parziale della blockchain usando un client BitTorrent scaricabile da SourceForge.

Esegui bitcoind in background con l'opzione -daemon:

```
$ bitcoind -daemon

Bitcoin version v0.9.0rc1-beta (2014-01-31 09:30:15 +0100)
Using OpenSSL version OpenSSL 1.0.1c 10 May 2012
Default data directory /home/bitcoin/.bitcoin
Using data directory /bitcoin/
Using at most 4 connections (1024 file descriptors available)
init message: Verifying wallet...
dbenv.open LogDir=/bitcoin/database ErrorFile=/bitcoin/db.log
Bound to [::]:8333
Bound to 0.0.0.0:8333
init message: Loading block index...
Opening LevelDB in /bitcoin/blocks/index
Opened LevelDB successfully
Opening LevelDB in /bitcoin/chainstate
Opened LevelDB successfully

[... altri messaggi d'avvio ...]
```

Utilizzando la API JSON-RPC di Bitcoin Core dalla Riga di Comando

Il Core Client Bitcoin implementa un'interfaccia JSON-RPC a cui si può accedere anche utilizzando il programma da riga di comando bitcoin-cli. La riga di comando (detta anche terminale o shell) ci permette di sperimentare interattivamente con tutte le potenzialità che sono anche disponibili attraverso l'API. Per cominciare, invoca il comando help per visualizzare una lista dei comandi bitcoin RPC disponibili:

```
$ bitcoin-cli help
addmultisigaddress nrequired ["key",...] ( "account" )
addnode "node" "add|remove|onetry"
backupwallet "destination"
createmultisig nrequired ["key",...]
createrawtransaction [{"txid":"id","vout":n},...] {"address":amount,...}
decoderawtransaction "hexstring"
decodescript "hex"
dumpprivkey "bitcoinaddress"
dumpwallet "filename"
getaccount "bitcoinaddress"
getaccountaddress "account"
getaddednodeinfo dns ( "node" )
getaddressesbyaccount "account"
getbalance ( "account" minconf )
getbestblockhash
getblock "hash" ( verbose )
getblockchaininfo
getblockcount
```

```
getblockhash index
getblocktemplate ( "jsonrequestobject" )
getconnectioncount
getdifficulty
getgenerate
gethashespersec
getinfo
getmininginfo
getnettotals
getnetworkhashps ( blocks height )
getnetworkinfo
getnewaddress ( "account" )
getpeerinfo
getrawchangeaddress
getrawmempool ( verbose )
getrawtransaction "txid" ( verbose )
getreceivedbyaccount "account" ( minconf )
getreceivedbyaddress "bitcoinaddress" ( minconf )
gettransaction "txid"
gettxout "txid" n ( includemempool )
gettxoutsetinfo
getunconfirmedbalance
getwalletinfo
getwork ( "data" )
help ( "command" )
importprivkey "bitcoinprivkey" ( "label" rescan )
importwallet "filename"
keypoolrefill ( newsize )
listaccounts ( minconf )
listaddressgroupings
listlockunspent
listreceivedbyaccount ( minconf includeempty )
listreceivedbyaddress ( minconf includeempty )
listsinceblock ( "blockhash" target-confirmations )
listtransactions ( "account" count from )
listunspent ( minconf maxconf  ["address",...] )
lockunspent unlock [{"txid":"txid","vout":n},...]
move "fromaccount" "toaccount" amount ( minconf "comment" )
ping
sendfrom "fromaccount" "tobitcoinaddress" amount ( minconf "comment" "comment-to" )
sendmany "fromaccount" {"address":amount,...} ( minconf "comment" )
sendrawtransaction "hexstring" ( allowhighfees )
sendtoaddress "bitcoinaddress" amount ( "comment" "comment-to" )
setaccount "bitcoinaddress" "account"
setgenerate generate ( genproclimit )
settxfee amount
signmessage "bitcoinaddress" "message"
signrawtransaction "hexstring" (
[{"txid":"id","vout":n,"scriptPubKey":"hex","redeemScript":"hex"},...]
["privatekey1",...] sighashtype )
stop
```

```
submitblock "hexdata" ( "jsonparametersobject" )
validateaddress "bitcoinaddress"
verifychain ( checklevel numblocks )
verifymessage "bitcoinaddress" "signature" "message"
walletlock
walletpassphrase "passphrase" timeout
walletpassphrasechange "oldpassphrase" "newpassphrase"
```

Ottenendo Informazioni sullo Stato del Client Bitcoin Core

Commands: getinfo

Il comando RPC di Bitcoin getinfo mostra informazioni base riguardo lo stato del nodo bitcoin del network , del wallet, e del database blockchain. Utilizza bitcoin-cli per eseguirlo:

```
$ bitcoin-cli getinfo
```

```
{
    "version" : 90000,
    "protocolversion" : 70002,
    "walletversion" : 60000,
    "balance" : 0.00000000,
    "blocks" : 286216,
    "timeoffset" : -72,
    "connections" : 4,
    "proxy" : "",
    "difficulty" : 2621404453.06461525,
    "testnet" : false,
    "keypoololdest" : 1374553827,
    "keypoolsize" : 101,
    "paytxfee" : 0.00000000,
    "errors" : ""
}
```

I dati sono ritornati in JavaScript Object Notation (JSON), un formato che può facilmente essere compreso da tutti i linguaggi di programmazione, ma che è anche abbastanza leggibile dall'utente. Oltre a questi dati vedremo i numeri di versione per il client (90000), del protocollo (70002), e infine del wallet (60000). Possiamo vedere il balance attuale contenuto nel wallet, che attualmente è zero. Possiamo notare la block height attuale, che mostra quanti blocchi sono presenti, secondo il client (286216). Possiamo inoltre notare varie statistiche riguardo il network bitcoin e le impostazioni relative a questo client. Esploreremo queste impostazioni in maggiore dettaglio nel corso di questo capitolo.

TIP

Ci vorrà un po' di tempo, probabilmente più di un giorno, prima che bitcoind "raggiunga" l'altezza della blockchain (block height) attuale man mano che scarica i blocchi da altri client bitcoin. Puoi controllare il progresso utilizzando getinfo per vedere il numero di blocchi conosciuti.

Setup del Wallet e Encryption (Protezione Crittografica)

Comandi: encryptwallet, walletpassphrase

Prima di procedere nel generare chiavi e altri comandi, dovresti per prima cosa criptare il wallet con una password. Per questo esempio, utilizzerai il comando encryptwallet con la password "foo". Ovviamente, sostituisci "foo" con una password più complessa e sicura!

```
$ bitcoin-cli encryptwallet foo
wallet criptato; Il server di bitcoin si fermerà, si riavvierà per eseguire il wallet
encrypted. La keypool è stata refreshata, dovrai effettuare un nuovo backup.
$
```

Puoi verificare che il wallet sia stato criptato lanciando nuovamente il comando getinfo. Questa volta noterai un nuovo elemento chiamato unlocked_until. Questo è un contatore che mostra per quanto tempo il processo di decriptazione della password del wallet sarà salvato in memoria, mantenendo il wallet aperto. All'inizio questo valore sarà impostato a zero, che sta a significare che il wallet è chiuso (protetto).

```
$ bitcoin-cli getinfo
```

```
{
    "version" : 90000,

#[... altre informazioni...]

    "unlocked_until" : 0,
    "errors" : ""
}
$
```

Per sbloccare il wallet, digita il comando walletpassphrase, che prende due parametri—la password ed un numero di secondi fino a quando il wallet sarà di nuovo bloccato automaticamente (un contatore):

```
$ bitcoin-cli walletpassphrase foo 360
$
```

Potrai confermare che il wallet è sbloccato e vedere il timeout eseguendo nuovamente getinfo:

```
$ bitcoin-cli getinfo
```

```
{
    "version" : 90000,

#[... altre informazioni ...]

    "unlocked_until" : 1392580909,
    "errors" : ""
}
```

Backup del Wallet, Dump Plain-text, e Restore

Comandi: backupwallet, importwallet, dumpwallet

A questo punto, faremo un po di pratica nel creare un file di backup del wallet e poi effettueremo il ripristino del wallet dal file di backup. Utilizza il comando backupwallet per effettuare il backup, fornendo come parametro il nome del file. In questo esempio effettuiamo il backup del wallet nel file *wallet.backup*:

```
$ bitcoin-cli backupwallet wallet.backup
$
```

A questo punto, esegui il ripristino del tuo file di backup, usa il comando importwallet. Se il tuo wallet è protetto (locked), dovrai prima effettuare l'unlock (vedi walletpassphrase nella sezione precedente) per importare il file di backup:

```
$ bitcoin-cli importwallet wallet.backup
$
```

Il comando dumpwallet può essere utilizzato per fare il dump del wallet in un file di testo che sia leggibile dall'utente (human-readable):

```
$ bitcoin-cli dumpwallet wallet.txt
$ more wallet.txt
# Wallet dump created by Bitcoin v0.9.0rc1-beta (2014-01-31 09:30:15 +0100)
# * Created on 2014-02- 8dT20:34:55Z
# * Best block at time of backup was 286234
(000000000000000f74f0bc9d3c186267bc45c7b91c49a0386538ac24c0d3a44),
#   mined on 2014-02- 8dT20:24:01Z

KzTg2wn6Z8s7ai5NA9MVX4vstHRsqP26QKJCzLg4JvFrp6mMaGB9 2013-07- 4dT04:30:27Z change=1 #
addr=16pJ6XkwSQv5ma5FSXMRPaXEYrENCEg47F
Kz3dVz7R6mUpXzdZy4gJEVZxXJwA15f198eVui4CUivXotzLBDKY 2013-07- 4dT04:30:27Z change=1 #
addr=17oJds8kaN8LP8kuAkWTco6ZM7BGXFC3gk
[... molte altre chiavi ...]

$
```

Indirizzi del Wallet e Ricezione delle Transazioni

Comandi: getnewaddress, getreceivedbyaddress, listtransactions, getaddressesbyaccount, getbalance

Il client di riferimento bitcoin mantiene un pool di indirizzi, la cui dimensione viene visualizzata da keypoolsize quando si usa il command getinfo. Questi indirizzi vengono generati automaticamente e possono quindi essere utilizzati come indirizzi di ricezione pubblici o come indirizzi di modifica. Per ottenere uno di questi indirizzi, utilizzare il comando: getnewaddress:

```
$ bitcoin-cli getnewaddress
1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL
```

Ora, possiamo utilizzare questo indirizzo per inviare una piccola somma di bitcoin al nostro wallet bitcoind da un wallet esterno (assumendo che tu abbia qualche bitcoin in un'exchange, wallet web, o un'altro wallet bitcoin da qualche parte). Per questo esempio, invieremo 50 millibit (0.050 bitcoin) all'indirizzo precedente.

Possiamo ora interrogare il client bitcoind per il valore ricevuto da questo indirizzo, e specificare quante conferme sono necessarie prima che il valore sia incluso nel balance. Per questo esempio, specificheremo zero conferme. Pochi secondi dopo aver inviato i bitcoin da un'altro wallet, vedremo il cambio di stato nel wallet. Utilizzeremo getreceivedbyaddress con il nuovo indirizzo e il numero di conferme impostato a zero (0):

```
$ bitcoin-cli getreceivedbyaddress 1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL 0
0.05000000
```

Se omettiamo lo zero dalla fine di questo comando, vedremo solo i valori che hanno almeno minconf conferme, dove minconf è l'impostazione per il numero minimo di conferme prima che una transazione vada a far parte del balance. L'impostazione minconf è specificata nel file di

configurazione bitcoind. Visto che la transazione che sta inviando questi bitcoin è trasmessa solo da pochi secondi, non sarà ancora stata confermata e quindi vedremo un balance di zero:

```
$ bitcoin-cli getreceivedbyaddress 1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL
0.00000000
```

Le transazioni ricevute dal wallet possono essere mostrate usando il comando listtransactions:

```
$ bitcoin-cli listtransactions
```

```
[
    {
        "account" : "",
        "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
        "category" : "receive",
        "amount" : 0.05000000,
        "confirmations" : 0,
        "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
        "time" : 1392660908,
        "timereceived" : 1392660908
    }
]
```

Possiamo listare tutti gli indirizzi presenti nel wallet usando il comando getaddressesbyaccount:

```
$ bitcoin-cli getaddressesbyaccount ""
```

```
[
    "1LQoTPYy1TyERbNV4zZbhEmgyfAipC6eqL",
    "17vrg8uwMQUibkvS2ECRX4zpcVJ78iFaZS",
    "1FvRHWhHBBZA8cGRRsGiAeqEzUmjJkJQWR",
    "1NVJK3JsL41BF1KyxrUyJW5XHjunjfp2jz",
    "14MZqqzCxjc99M5ipsQSRfieT7qPZcM7Df",
    "1BhrGvtKFjTAhGdPGbrEwP3xvFjkJBuFCa",
    "15nem8CX91XtQE8B1Hdv97jE8X44H3DQMT",
    "1Q3q6taTsUiv3mMemEuQQJ9sGLEGaSjo81",
    "1HoSiTg8sb16oE6SrmazQEwcGEv8obv9ns",
    "13fE8BGhBvnoy68yZKuWJ2hheYKovSDjqM",
    "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
    "1KHUmVfCJteJ21LmRXHSpPoe23rXKifAb2",
    "1LqJZz1D9yHxG4cLkdujnqG5jNNGmPeAMD"
]
```

Infine, il comando getbalance mostrerà il saldo totale del wallet, sommando tutte le transazioni confermate con almeno minconf conferme:

```
$ bitcoin-cli getbalance
0.05000000
```

TIP	Se la transazione non è ancora confermata, il saldo ritornato da getbalance sarà zero. L'opzione di configurazione "minconf" determina il numero minimo di conferme necessarie prima che una transazione venga inclusa nel saldo.

Esplorando e Decodificando le Transazioni

Comandi: gettransaction, getrawtransaction, decoderawtransaction

Esploreremo la transazione in entrata che è stata listata precedentemente utilizzando il comando gettransaction. Possiamo recuperare una transazione tramite il suo hash di transazione, mostrato precedentemente come txid, con il comando gettransaction:

```
{
    "amount" : 0.05000000,
    "confirmations" : 0,
    "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
    "time" : 1392660908,
    "timereceived" : 1392660908,
    "details" : [
        {
            "account" : "",
            "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
            "category" : "receive",
            "amount" : 0.05000000
        }
    ]
}
```

TIP	Gli ID di transazione non sono autoritativi fino a che una transazione non è confermata. L'assenza di un hash di transazione nella blokchain non significa che la transazione non sia stata processata. Questo è conosciuto come "transaction malleability," malleabilità di transazione, perchè gli hash di transazione possono essere modificate prima di essere confermati in un blocco. Dopo essere stati confermati, i txid sono immutabili e autoritativi.

La rappresentazione di una transazione come mostrata dal comando gettransaction è nella forma semplificata. Per ottenere il codice di transazione completo e decodificarlo, utilizzeremo due comandi: getrawtransaction e decoderawtransaction. Per primo, getrawtransaction prende come parametro l'*hash di transazione (txid)* e ritorna la transazione completa come stringa "raw" esadecimale, esattamente nella stessa forma come quella esistente sul network bitcoin:

Per decodificare questa stringa esadecimale, utilizziamo il comando decoderawtransaction. Copia e incolla l'esadecimale come primo parametro di decoderawtransaction per ottenere il contenuto

completo interpretato come struttura dati JSON (per ragioni di formattazione la stringa esadecimale del seguente esempio è stata troncata):

La decodifica della transazione mostra tutte le parti di questa transazione, inclusi gli input e gli output. In questo caso possiamo notare che la transazione che ha accreditato il nostro indirizzo con 50 millibit ha usato un input e ha generato due output. L'input di questa transazione era l'output di una transazione precedentemente confermata (mostrato come "vin txid" che inizia con d3c7). I due output corrispondono al valore di 50 millibit e ad un output contenente il resto reinviato al mittente.

Possiamo esplorare ulteriormente la blockchain esaminando la transazione precedente referenziata dal suo txid in questa transazione usando lo stesso comando (i.e., gettransaction). Saltando da transazione a transazione possiamo seguire una catena di transazioni precedenti fino a che i bitcoin non siano stati trasmessi da un indirizzo di un proprietario a un'altro.

Una volta che la transazione che abbiamo ricevuto è stata confermata e inclusa in un blocco il comando gettransaction ritornerà informazioni addizionali, mostrando l'*identificatore dell'hash di blocco* nella quale l'hash di transazione è stato incluso:

Qui, possiamo vedere la nuova informazione nelle voci denominate blockhash (l'hash del blocco in cui la transazione è stata inclusa), e blockindex con il valore 18 (indicando che la nostra transazione è stata la 18esima transazione nel determinato blocco).

Indice del Database delle Transazioni e Opzione txindex

Di base, Bitcoin Core, costruisce un database contenente *solamente* le transazioni relative al wallet dell'utente. Se vuoi essere capace di accedere ad *ogni* transazione con comandi come gettransaction, avrai bisogno di configurare Bitcoin Core per costruire un indice delle transazioni completo, che può essere ottenuto attraverso l' opzione txindex. Imposta txindex=1 nel file di configurazione di Bitcoin Core (di solito trovato nella tua home directory in *.bitcoin/bitcoin.conf*). Una volta cambiato questo parametro, dovrai riavviare bitcoind e aspettare perchè ricostruisca l'indice.

Esplorando i Blocchi

Comandi: getblock, getblockhash

Ora che sappiamo in quale blocco la nostra transazione è stata inclusa, possiamo interrogare quel blocco. Utilizzeremo il comando getblock con l'hash del blocco come parametro:

Il blocco contiene 367 transazioni e come potete vedere, la 18esima transazione listata (9ca8f9...) è il txid di quella che ha accreditato 50 millibit al nostro indirizzo. La voce height ci dice che questo è il 286384esimo blocco della blockchain.

Possiamo inoltre recuperare un blocco tramite la sua block height usando il comando getblockhash, il quale richiede la block height come parametro e ritorna il block hash corrispondente per quel blocco:

Qui, consultiamo il block hash del "genesis block", il primo blocco estratto da Satoshi Nakamoto, a

quota zero. Consultare questo blocco mostra:

I comandi getblock, getblockhash, e gettransaction possono essere usati per esplorare il database-blockchain, programmaticamente.

Creando, Firmando e Trasmettendo Transazioni Basate sul passaggio: [<phrase role="keep-together">Unspent Outputs</phrase>]

Comandi: listunspent, gettxout, createrawtransaction, decoderawtransaction, signrawtransaction, sendrawtransaction

Le transazioni di Bitcoin si basano sul concetto di "output" di spesa, che sono il risultato di transazioni precedenti, per creare una catena di transazioni che trasferisce la proprietà da un indirizzo all'altro. Il nostro portafoglio ha ora ricevuto una transazione che ha assegnato uno di questi output al nostro indirizzo. Una volta confermato, possiamo spendere quell'output.

All'inizio, usiamo il comando listunspent per mostrare tutti gli output non spesi *confermati* presenti nel nostro wallet:

```
$ bitcoin-cli listunspent
```

Possiamo notare che la transazione 9ca8f9... creata con un output (con vout con indice 0) assegnata all'indirizzo 1hvzSo... per l'amount di 50 millibit, che a questo punto ha ricevuto sette conferme. Le transazioni utilizzano output creati precedentemente come loro input referenziandoli da indici precedenti txid e vout. Creeremo una transazione che spenderà lo 0esimo vout del txid 9ca8f9... come suo input e lo assegnerà a un nuovo output che invierà valore al nuovo indirizzo.

Per iniziare, osserviamo l'output con maggiore dettaglio. Usiamo il comando gettxout per ottenere i dettagli di questo unspent output. Gli output di transazione sono sempre referenziati da un txid e un vout, e questi sono i parametri che passiamo a gettxout:

Quello che possiamo vedere qui è l'output che ha assegnato 50 millibit al nostro indirizzo 1hvz.... Per spendere questo output dobbiamo creare una nuova transazione. Per prima cosa, creiamo un'indirizzo a cui invieremo il denaro:

```
$ bitcoin-cli getnewaddress
1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb
```

Invieremo 25 millibit al nuovo indirizzo 1LnfTn... che abbiamo appena creato nel nostro wallet. Nella nostra nuova transazione, spenderemo l'output da 50 millibit e invieremo 25 millibit a questo nuovo indirizzo. Visto che dobbiamo spendere l'output *completamente* dalla transazione precedente, dobbiamo anche generare del resto. Genereremo un resto indietro verso l'indirizzo 1hvz..., inviando il resto indietro all'indirizzo da cui originava il valore. Infine, dovremmo inoltre pagare una fee per questa transazione. Per pagare la fee, dovremmo ridurre l'output del resto di 0.5 millibit, e ritornare 24.5 millibit in resto. La differenza tra la somma dei nuovi output (25 mBTC + 24.5 mBTC = 49.5 mBTC) e l'input (50 mBTC) sarà riscosso come fee dai miner.

Utilizzeremo il comando createrawtransaction per creare questa transazione. Come parametro di createrawtransaction forniremo l'input di transazione (l'unspent output di 50 millibit dalla nostra transazione confermata) e i due output di transazione (i bitcoin inviati al nuovo indirizzo e il resto che è stato re-inviato indietro all'indirizzo precedente):

Il comando createrawtransaction produce una stringa raw esadecimale che codifica i dettagli della transazione che abbiamo fornito. Confermiamo che tutto sia corretto decodificando questa stringa raw utilizzando il comando decoderawtransaction:

Sembra corretto! La nostra transazione "consuma" l'unspent output dalla nostra transazione confermata e successivamente lo spende in due output, uno da 25 millibit verso il nostro nuovo indirizzo e uno da 24.5 millibit come resto indietro al nostro indirizzo originario. La differenza di 0.5 millibit rappresenta la transaction fee e sarà accreditata al miner che troverà il blocco che include la nostra transazione.

Come potrai notare, la transazione contiene uno scriptSig vuoto visto che non l'abbiamo ancora firmato. Senza una firma, questa transazione non ha senso; non abbiamo avuto la prova che *siamo proprietari* dell'indirizzo dal quale l'unspent output proviene. Firmando, rimuoviamo il blocco sull'output e proviamo che siamo noi proprietari di esso e che possiamo spendere l'output. Necessita della stringa di transazione in esadecimale, come parametro:

TIP	Un wallet criptato deve essere sbloccato prima di poter firmare una transazione perchè l'azione di firmare richiede l'accesso alle chiavi private nel wallet.

Il comando signrawtransaction ritorna un'altra transazione raw con codifica esadecimale. La decodificheremo per vedere cosa è stato modificato, con il comando decoderawtransaction:

A questo punto, gli input usati nella transazione contengono uno scriptSig, che è una firma digitale che prova la proprietà dell'indirizzo 1hvz... e che rimuoverà il blocco (lock) sull'output di modo che potrà essere speso. La firma rende questa transazione verificabile da ogni nodo nel network bitcoin.

Adesso è tempo di inviare la nuova transazione appena creata al network. Possiamo fare questo con il comando sendrawtransaction, che prende la stringa raw esadecimale prodotta da signrawtransaction. Questa è la stessa stringa che abbiamo appena decodificato:

Il comando sendrawtransaction ritorna un *hash di transazione* (transaction hash - txid) non appena invia la transazione al network. Possiamo a questo punto interrogare quell'ID di transazione con gettransaction:

```
{
    "amount" : 0.00000000,
    "fee" : -0.00050000,
    "confirmations" : 0,
    "txid" : "ae74538baa914f3799081ba78429d5d84f36a0127438e9f721dff584ac17b346",
    "time" : 1392666702,
    "timereceived" : 1392666702,
    "details" : [
        {
            "account" : "",
            "address" : "1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb",
            "category" : "send",
            "amount" : -0.02500000,
            "fee" : -0.00050000
        },
        {
            "account" : "",
            "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
            "category" : "send",
            "amount" : -0.02450000,
            "fee" : -0.00050000
        },
        {
            "account" : "",
            "address" : "1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb",
            "category" : "receive",
            "amount" : 0.02500000
        },
        {
            "account" : "",
            "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
            "category" : "receive",
            "amount" : 0.02450000
        }
    ]
}
```


Come in precedenza, possiamo inoltre esaminare questo con maggiore dettaglio utilizzando i comandi getrawtransaction e decodetransaction. Questi comandi ritorneranno la medesima stringa esadecimale che abbiamo prodotto e decodificato precedentemente prima di inviarla sul newtork.

Client Alternativi, Librerie, e Toolkits

Oltre il client di riferimento (bitcoind), altri client e librerie possono essere utilizzati per interagire con il network bitcoin e le strutture dati. Queste sono implementate in una varietà di linguaggi di programmazione, offrendo ai programmatori interfacce native per il loro linguaggio.

Implementazioni alternativi includono:

libbitcoin

Development Toolkit Bitcoin C++ Cross-Platform

bitcoin explorer

Strumento Bitcoin da Riga di Comando

bitcoin server

Full Node Bitcoin e Query Server

bitcoinj

Una libreria client bitcoin full-node in Java

btcd

Una libreria client bitcoin full-node in Go

Bits of Proof (BOP)

Una implementazione di bitcoin enterprise-level

picocoin

Un'implementazione in C di una libreria client leggera per bitcoin

pybitcointools

Una libreria bitcoin in Python

pycoin

Un'altra libreria bitcoin in Python

Esistono molte altre librerie scritte in una gran varietà di altri linguaggi di programmazione comuni e molte altre col tempo saranno create.

Libbitcoin e Bitcoin Explorer

La libreria libbitcoin è un kit di sviluppo multi plattaforma in C++ che supporta il full node libbitcoin-server e il tool da riga di comando Bitcoin Explorer (bx).

I comandi di bx offrono molte funzionalità medesime a quelle di bitcoind, e comandi simili a quelli illustrati in questo capitolo. I comandi di bx inoltre offrono anche un numero limitato di strumenti per la gestione delle chiavi e per la loro manipolazione, che non sono presenti in bitcoind, inclusi chiavi deterministiche type-2 e encoding di chiavi mnemoniche, oltre che a stealth address, pagamenti e supporto per eseguire query.

Installare Bitcoin Explorer

Per utilizzare Bitcoin Explorer, semplicemente scarica l'eseguibile firmato per il tuo sistema operativo. Le build sono disponibili per mainnet e testnet, per Linux, OS X, e Windows.

Digita bx senza alcun parametro per mostrare la lista di tutti i comandi disponibili (vedi [appdx_bx]).

Bitcoin Explorer inoltre fornisce un installer per con la possibilità di compilare il progetto da sorgente su Linux e OS X, e anche un progetto Visual Studio per Windows. I sorgenti possono anche essere compilati manualmente usando Autotools. In questo modo verrà installata anche la libreria libbitcoin, che è una delle dipendenze.

TIP	Bitcoin Explorer offre molti altri comandi per codificare e decodificare indirizzi, e convertirli in e da differenti formati e rappresentazioni. Usali per esplorare i vari formati come Base16 (hex), Base58, Base58Check, Base64, etc.

Installare Libbitcoin

La libreria libbitcoin fornisce un installer per compilare da sorgente per Linux e OS X, e anche un progetto Visual Studio per Windows. I sorgenti possono anche essere compilati manualmente utilizzando Autotools.

TIP	L'installer di Bitcoin Explorer installer installa sia bx che la libreria libbitcoin, quindi se hai già compilato bx da sorgente, puoi saltare questo passaggio.

pycoin

La libreria Python *pycoin*, originariamente scritta e mantenuta da Richard Kiss, è una libreria basata sul linguaggio Python che supporta la manipolazione di chiavi bitcoin e di transazioni, supportando anche a un buon livello il linguaggio di scripting bitcoin, abbastanza per poter gestire transazioni non-standard.

La libreria pycoin supporta sia Python 2 (2.7.x) e Python 3 (successivo alla 3.3); e viene fornito con utilità da riga di comando molto utili, ku e tx. Per installare pycoin 0.42 con Python 3 in un virtual environment (venv), usa il seguente comando:

```
$ python3 -m venv /tmp/pycoin
$ . /tmp/pycoin/bin/activate
$ pip install pycoin==0.42
Scaricando/scompattando pycoin==0.42
  Downloading pycoin-0.42.tar.gz (66kB): 66kB downloaded
  Lanciando setup.py (path:/tmp/pycoin/build/pycoin/setup.py) egg_info per il
pacchetto pycoin

Installing collected packages: pycoin
  Running setup.py install for pycoin

    Installing tx script to /tmp/pycoin/bin
    Installing cache_tx script to /tmp/pycoin/bin
    Installing bu script to /tmp/pycoin/bin
    Installing fetch_unspent script to /tmp/pycoin/bin
    Installing block script to /tmp/pycoin/bin
    Installing spend script to /tmp/pycoin/bin
    Installing ku script to /tmp/pycoin/bin
    Installing genwallet script to /tmp/pycoin/bin
Successfully installed pycoin
Cleaning up...
$
```

Qui c’è uno script Python di esempio per prendere e spendere un po di bitcoin usando la libreria pycoin:

```
#!/usr/bin/env python

from pycoin.key import Key

from pycoin.key.validate import is_address_valid, is_wif_valid
from pycoin.services import spendables_for_address
from pycoin.tx.tx_utils import create_signed_tx

def get_address(which):
    while 1:
        print("enter the %s address=> " % which, end='')
        address = input()
        is_valid = is_address_valid(address)
        if is_valid:
            return address
        print("invalid address, please try again")

src_address = get_address("source")
spendables = spendables_for_address(src_address)
print(spendables)

while 1:
    print("enter the WIF for %s=> " % src_address, end='')
    wif = input()
    is_valid = is_wif_valid(wif)
    if is_valid:
        break
    print("invalid wif, please try again")

key = Key.from_text(wif)
if src_address not in (key.address(use_uncompressed=False),
key.address(use_uncompressed=True)):
    print("** WIF doesn't correspond to %s" % src_address)
print("The secret exponent is %d" % key.secret_exponent())

dst_address = get_address("destination")

tx = create_signed_tx(spendables, payables=[dst_address], wifs=[wif])

print("here is the signed output transaction")
print(tx.as_hex())
```

Per esempi che usano le utilità da riga-di-comando ku e tx, vedi [appdxbitcoinimpproposals].

btcd

btcd è un'implementazione di un full-node bitcoin scritta in Go. Attualmente scarica, valida e funge da blockchain server usando le stesse regole (bug inclusi) per l'accettazione di blocchi come

l'implementazione di riferimento, bitcoind. Inoltre trasmette correttamente nuovi blocchi confermati, mantiene una transaction pool, e trasmette transazioni singole che non sono ancora facenti parte di un blocco. Assicura che tutte le transazioni singole ammesse nella pool seguano le regole necessarie e inoltre include la stragrande maggioranza di controlli più rigorosi che filtrano le transazioni basate sui requisiti dei miner (transazioni "standard").

Una differenza chiave tra btcd e bitcoind è quella che btcd non include funzionalità wallet, e questo è stata una decisione intenzionale sul suo design. Questo significa che non puoi effettuare o ricevere pagamenti direttamente con btcd. Quella funzionalità è fornita dai progetti btcwallet e btcgui, i quali sono entrambi in attivo sviluppo. Altre differenze degne di nota tra btcd e bitcoind includono il supporto di btcd sia per richieste HTTP POST (come bitcoind) e quello consigliato Websockets, e il fatto che le connessioni RPC di btcd hanno di base, TLS abilitato.

Installare btcd

Per installare btcd per Windows, scarica e esegui il file msi disponibile in questo repository GitHub, o esegui il seguente comando su Linux, assumento che tu hai già installato il linguaggio Go:

```
$ go get github.com/conformal/btcd/...
```

Per aggiornare btcd all'ultima versione, basta eseguire:

```
$ go get -u -v github.com/conformal/btcd/...
```

Controllare btcd

btcd ha un certo numero di opzioni di configurazioni, che potrai vedere eseguendo:

```
$ btcd --help
```

btcd è fornito insieme a alcune utilità come btcctl, che è una utility da riga di comando che può essere usata sia per controllare che per interrogare btcd attraverso RPC. btcd non abilita il suo server RPC al normale avvio; devi configurare almeno un username e una password RPC nei seguenti file di configurazione:

- *btcd.conf*:

```
[Opzioni dell'Applicazione]
rpcuser=myuser
rpcpass=SomeDecentp4ssw0rd
```

- *btcctl.conf*:

```
[Opzioni dell'Applicazione]
rpcuser=myuser
rpcpass=SomeDecentp4ssw0rd
```

O se vuoi sovrascrivere i file di configurazione dalla riga di comando:

```
$ btcd -u myuser -P SomeDecentp4ssw0rd
$ btcctl -u myuser -P SomeDecentp4ssw0rd
```

Per una lista di opzioni disponibili, esegui il seguente comando:

```
$ btcctl --help
```